Systems programming

6 – pipes

MEEC LEEC MEAer LEAer MEIC-A

João Nuno Silva

TÉCNICO LISBOA

Bibliography

- Pipes - Summary, M. D. McIlroy
- The Linux Programming Interface, Michael Kerrisk
  - Chapter 44
- Beej's Guide to Interprocess Communication, Brian Hall
  - Chapters 4-5
- The Linux Programmer's Guide - Sven Goldt, Sven van der Meer, Scott Burke
  - Sections 6.1 .. 6.3

- M. D. McIlroy - October 11, 1964
  - We should have some ways of coupling programs like garden hose
  - screw in another segment when it becomes necessary to message data in another way
  - This is the way of IO also.
- In Unix pipes are the original inter-process communication mechanisms
  - They are anonymous and allow related processes to exchange data
  - It is also deployed by the shell
  - Uses simple communication protocol (just text)

Pipes in the shell

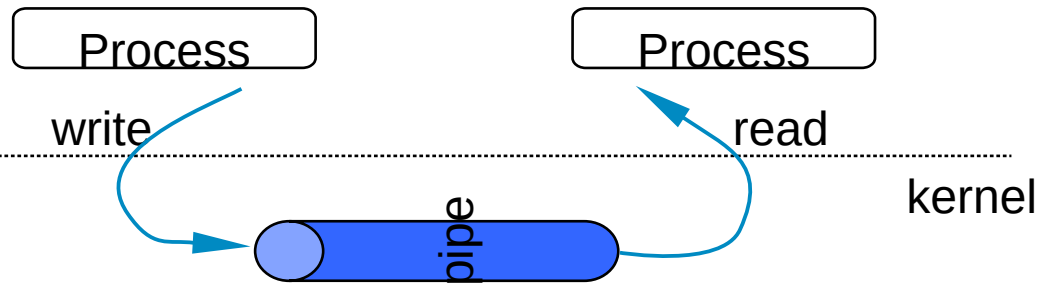
- pipe operation
  - connects the standard output of one program to the standard input of another.
  - A chain of programs connected in this way is called a pipeline.
- `ls | wc`
  - Counts character/word/line count for the current directory listing
- `tr -c '[:alnum:]' '[\n*]' | sort -iu | grep -v '^[0-9]*$'`

Pipes in the shell

- All the stages in a pipeline run concurrently.
  - Each stage waits for input on the output of the previous one,
  - no stage has to exit before the next can run.
- It is unidirectional.
  - p1 | p2 p1 → p2 p1 ← p2
  - Impossible to pass information back
    - Just p2 exit notification

Pipes

- Read/Writes
  - File operations
- Processes
  - Should be related
    - Father/soon
    - Brothers



Pipes

- Pipe creation:

```
int pipe(int fd[2]);
int pipe2(int pipefd[2],
          int flags);
```

- Opens two files
  - **`fd[0]`** descriptor open for reading
  - **`fd[1]`** descriptor open for writing

- Returns
  - 0 successful
  - 1 unsuccessful
-
- Pipes can only
  - connect processes with a common ancestor
- Pipe
  - is managed as a open file

Pipes

- Communication (data read/write)
  - **`ssize_t read(int fd, void *buf, size_t count);`**
  - **`ssize_t write(int fd, void *buf, size_t count);`**
- 1st argument
  - File descriptor (fd[0] or fd[1])
- 2nd argument
  - data buffer address (data destination/source)
- 3rd argument
  - number of bytes to read/write
- Return number of bytes read/written

Closing Pipes

- Closing all write ends
  - Read will return 0
- Closing all read ends
  - Write will produce an exception
- Processes should close not needed ends
  - For previous notifications to work

Pipes

- Pipes offer some primitive synchronization
  - Process wait for the others
- If a process attempts to read from an empty pipe,
  - then **`read`** will block until data is available.
- If a process attempts to write to a full pipe,
  - then **`write`** blocks until sufficient data has been read from the pipe to allow the write to complete.

Pipes

- The communication channel provided is
  - byte stream:
- there is no concept of message boundaries.
- Reads returns all available data up to the size of buffer
  - **write(10 bytes) → write(10 bytes) → read (13 bytes)**
  - Is possible, leaves 7 bytes in the pipe

Pipes

- Nonblocking I/O is possible
  - using O_NONBLOCK status flag
  - reads/writes return immediately
    - Programmer must verify if data was transferred
- Writes on pipe are atomic:
  - the output data is written to the pipe as a contiguous sequence.
  - Even if two writes are concurrent
    - In different processes

Pipes

- A pipe has a limited capacity.
  - Data is stored in the operating system
    - Until read, pip destroyed or reboot
  - Different OS have different limits for the pipe capacity
    - POSIX.1-2001 requires PIPE_BUF to be at least 512 bytes.
    - On Linux PIPE_BUF is 4096 bytes.
    - Applications should not rely on a particular capacity
    - application should consume data as soon as possible

Pipes

- Messages are limited to byte streams.
- Information flow is unidirectional
  - One process reads one process writes
  - Uses file descriptors functionality
- Major limitation
  - Processes should be related
- How to implement pipes that are accessible by other processes?
  - Giving them a name
  - Registering them in the File system

Named pipes / FIFOs

FIFO / Named Pipes

- To solve Pipes limitations
  - FIFOs were defined
    - Also referred as named pipes
- Can be used by unrelated processes
- Are referred and identified by a file in the file system
- A FIFO is special file similar to a pipe,
  - That is created in a different way
  - Instead of being an anonymous communications channel,
    - FIFO is entered into the file system by calling mkfifo()
  -

FIFO / Named Pipes

- FIFO creations
  - int mkfifo(const char *pathname, mode_t mode);
  - 1st argument - FIFO name (full path)
  - 2nd argument - Access permissions (like a regular file)
- On success **mkfifo()** returns 0.
  - In the case of an error, -1 is returned (errno is set appropriately).
- Once a FIFO is created a special file is available
  - any process can open it for reading or writing, in the same way as an ordinary file.

FIFO / Named Pipes

- Before being used the FIFO should be opened

  int open(const char *pathname, int flags);
  - Like a regular file
- 1st argument - FIFO name
- 2nd argument - Bits that define access mode
  - O_RDONLY (just reading)
  - O_RDONLY (just writing)
  - O_NONBLOCK (non blocking I/O)
- The return value is
  - -1 in case of error
  - Or a positive file descriptor

FIFO / Named Pipes

- A FIFO has to be opened at both ends simultaneously
  - before any process returns from open
  - Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa.
- After opening, FIFO offer the same synchronization as pipes
  - empty/full FIFO
- Data access is similar to a pipe
  - Data stream and atomicity

Next on PSIS

- sockets